Code Generator for Register Optimizations on GPUs

ACM Reference format:
. 2016. Code Generator for Register Optimizations on GPUs. 1, 1, Article 1 (January 2016), 2 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 GRAMMAR

The code generator presented here accepts as input a sequence of stencil statements that are expressed in a restricted subset of C consistent with the grammar described in figure 1. The input to the code generator is a full stencil code in a .cu file, with the regions that the user wants to reorder annotated with `#pragma begin stencil...` and a matching `#pragma end`. The parser is simplified and does not perform any syntactic verification, so when creating a new benchmark, it is the user's responsibility to ensure that the syntax adheres to the specifications for the code generator to work correctly.

We impose the following restrictions on the statements:

- The statements must not modify the iterators (e.g., i, j and k in listing 1) in any way.
- The statements can only have mathematical functions like sine, sqrt, cosine, etc. on the RHS, that are side-effect free.
- The accessed locations in the stencil statements are either scalars or array elements. The array index expressions can only comprise the loop iterators, integers or simple arithmetic operations between them (+,-,*,/). Currently, the code generator only handles simpler arithmetic for multiplication and division – the finite state machine (FSM) inside the code generator must be able to separate the result of the arithmetic operation into *one* arithmetic operation over iterators, and an integer value. For example, we allow A[k+1+1][2*j+2][i], but not something as complex as A[(k+2)*(k+3)][j][i], since this access will result in (k^2+5k) and 6, where (k^2+5k) is not a single arithmetic operation on the iterators.

In the grammar of figure 1, a *BaseExpr* in a statement represents an accessed storage location, and is identified by a *label*. The code generator tries to ensure that all the accesses to the same storage location within a statement have the same label. The labels are generated by the FSM, and it uses a cannonical representation for the array indices so that `A[i+1+1]` is assigned the same label as `A[i+2]`. The above mentioned restrictions allow the FSM to easily cannonicalize the accesses, and check for the equivalence of two labels. Across statements, the accesses to a storage location M will be identified by the same label if there is no write to M in between their execution. Once the parsing is complete, the code generator lowers the input further to sub-statements consistent with the grammar of figure 2. Informally, this lowering is like GIMPLE IR of GCC with accumulations.

2 CREATING AN EXAMPLE

Listing 1 shows a snippet of a cuda input code that can be passed as an input to the reordering framework. The computation in lines 5-7 is affine, and in compliance with the restrictions we placed on the input grammar. Note the pragmas in line 4 and 8 that surround the statement – these demarcate the region that the code generator can parse and optimize. Additionally, the pragma supplies the unrolling factor along each dimension. Given an input file with several such pragma-demarcated regions, the steps in optimizing the code will be:

2016. XXXX-XXXX/2016/1-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

⟨Program⟩ :: ⟨StmtList⟩;
⟨StmtList⟩ :: ⟨Stmt⟩⟨StmtList⟩ | ⟨Stmt⟩
⟨Stmt⟩ :: ⟨BaseExpr⟩⟨AssignOp⟩⟨Expr⟩;
⟨Expr⟩ :: Side-effect-free expression of ⟨BaseExpr⟩
⟨BaseExpr⟩ :: ⟨ID⟩ | ⟨ArrayExpr⟩
⟨ArrayExpr⟩ :: ⟨ID⟩⟨ArrayDims⟩
⟨ArrayDims⟩ :: [⟨AccessExpr⟩]⟨ArrayDims⟩ | [⟨AccessExpr⟩]
⟨AccessExpr⟩ :: Side-effect-free expression of ⟨AccessElement⟩
⟨AccessElement⟩ :: ⟨Iterator⟩ | ⟨Const⟩
⟨AssignOp⟩ :: = | += | -= | *= | ...
⟨DataType⟩ :: double | float | int | bool | ...

Fig. 1. Grammar for valid input statements to the code generator

⟨CodeBody⟩ :: ⟨StmtList$_l$⟩;
⟨StmtList$_l$⟩ :: ⟨Assignment$_l$⟩;⟨StmtList$_l$⟩ | ⟨Assignment$_l$⟩
⟨Assignment$_l$⟩ :: ⟨BaseExpr⟩⟨AssignOp⟩⟨Expr$_l$⟩
⟨Expr$_l$⟩ :: ⟨BaseExpr⟩⟨BinaryOp$_l$⟩⟨BaseExpr⟩ | ⟨BaseExpr⟩
⟨BinaryOp$_l$⟩ :: + | - | * | / | | | & | ...

Fig. 2. Grammar for the lowered 3-address IR for instruction reordering

Listing 1. The input representation with pragmas

```
1   for (k=1; k<=N-2; k++) {
2       for (j=1; j<=N-2; j++) {
3           for (i=1; i<=N-2; i++) {
4 #pragma begin stencil1 unroll k=4,j=1,i=1
5                   out[k][j][i] = a*(in[k+1][j][i]) + b*(in[k][j-1][i] +
6                                   in[k][j][i-1] + in[k][j][i] + in[k][j][i+1] +
7                                   in[k][j+1][i]) + c*(in[k-1][j][i]);
8 #pragma end stencil1
9           }
10      }
11  }
```

(1) A script will remove the statements between all the pragma, and put them into separate `.idsl` files, each named based on the stencil name (e.g., `stencil1.idsl` for the example in listing 1). Therefore, each stencil name must be distinct.
(2) From the pool of `stencilname.idsl` files generated, the code generator will pick one at a time, parse it, optimize it, and write the results back in `stencilname.cu`.
(3) A script will then read the contents from `stencilname.cu`, and replace the original statements demarcated by `#pragma begin stencilname ... #pragma end stencilname` with the optimized code.

The changes are made in-place to the input cuda file. Please look at the benchmarks in the examples folder to learn more about how to use this code generator for your applications.